Incorporating Animation in Stepwise Development of Formal Specification *

Atif Mashkoor, Jean-Pierre Jacquot
LORIA, DEDALE Team, Nancy Université
Vandoeuvre-Lès-Nancy, France
{firstName.lastName}@loria.fr

Abstract

This paper explores the possibility to incorporate validation of formal specifications into their step-wise development process. The key idea in formal methods to assess that an implementation is correct is to break the verification into smaller proofs associated with each refinement step. Likewise, the technique of animation could be used with each refinement step to break its validation into smaller assessments. Animating an abstract specification often requires to alter it in ways that proof obligations cannot be discharged anymore. So, we have developed a process and a set of transformation rules whose application produce an animatable specification which may be non-provable, but which is guaranteed to have the same behavior. 10 rules have been identified; they are presented and discussed with a special emphasis on their validity. We relate how step-wise animation is used in two case studies and what we gain from this.

1 Introduction

Despite decades of advocacy, success in safety critical systems development, easier to read formal languages, and good proof tools, the use of formal specifications in software development is still not popular.

In the same time, we have seen the rise of model-oriented methods, i.e., graphical formalisms, which are widely practiced. While some, like SCADE [15], have a strong semantics and formal basis, most are less well defined. One of the reasons of the appeal of graphical formalisms is that users can be associated earlier in the development process. In particular, they can validate developers' understanding of the problem and early decisions.

Formal languages are notoriously difficult to read for the non-initiated. Furthermore, well written specifications often introduce abstract objects and operations that have no intuitive concrete counterpart. Hence, validation has to wait. This implies that the development of the specification requires an uncomfortable level of trust.

This difficulty has been identified long ago [4] along with a solution: providing a graphical animation of the specification. Tools have been provided to help visualize requirements and system specifications [5, 11, 20, 17, 19]. The question is then *When* can we begin validation?

Verification raises a similar question. In test-based verification procedures, we need to wait until actual piece of code is implemented and running. As the cost of correcting errors or misunderstandings in requirements increases dramatically during the development life-cycle, it makes a lot of sense to verify, and validate, as early as possible.

The pivotal concept of formal methods such as B [1] is the notion of refinement and its relation to correctness. The assessment of the correctness of a piece of code, its verification, is no more a unique big process step but it is broken down into small pieces along with the whole development process. The proof of correctness is the sum of the proofs of small assertions (invariant preservation, well-formedness, existence of abstraction function, etc.) associated to each refinement. Problems are then detected early. While a formal refinement process does not preclude a testing activity, the latter will be more focused on finding true implementation errors, not requirement problems.

Our aim is to introduce validation into refinement based processes. We expect to gain on two levels. First, early detection of problems in the requirements (say, misunderstanding about a certain behavior) should be easier and inexpensive to correct. Second, users can be involved into the development right from the start.

In this work, we focus on the "execution" of a specification as a mean to validate it. Thanks to the development of tools like Brama[1] or ProB [12][2], it is possible to ani-

*This work has been partially supported by the ANR (National Research Agency) in the context of the TACOS project, whose reference number is ANR-06-SETI-017 (http://tacos.loria.fr), and by the Pôle de Compétitivité Alsace/Franche-Comté in the context of the CRISTAL project (http://www.projet-cristal.org).

[1] http://www.brama.fr
[2] http://www.stups.uni-duesseldorf.de/ProB

mate specifications in B or Event-B [2] before they reach an implementation stage. However, there are restrictions on the kind of specifications that can be animated. As anticipated, non-constructive definitions, infinite sets, or complex quantified logic expressions are among the list of restrictions. Unfortunately well-written specifications often use these traits. Indeed, it is even advised that early specifications be highly abstract and non constructive.

When toying with Brama, we observed that small alterations to a specification often allowed us to animate it, but at the expense of loosing some of its formal properties: some proof obligations could not be discharged anymore. However, both specifications clearly described a common set of behaviors.

Those observations lead us to develop a technique to animate abstract specifications by systematic transformations. The product of the transformations is a specification which may be non provable, but which is guaranteed to have the same behavior as the formally correct initial specification. This goal is achieved through the design of a set of transformation heuristics whose correction is rigourously asserted and a rigorous process.

The presentation of the paper is organized as follows: Next section presents the language and tool we use: Event-B and Brama. Then we present the animation process and the transformation rules. Two case-studies are described thereafter to show how to employ the heuristics. Finally, we conclude with questions raised by this work and what should now be completed to have a technique that could be used as standard practice.

2 Tools and concepts

2.1 Event-B

Event-B is a formal language for modeling and reasoning about large reactive and distributed systems. Event-B is provided with tool support in the form of a platform for writing and proving specifications called RODIN[3].

Abstract Specification An Event-B abstract specification is encapsulated into a MODEL identified by a unique name. The system variables are given in the VARIABLES part. An INVARIANT defines the state space of the variables and their safety properties. Each event in the EVENTS part is a substitution statement. Their semantic is given by the weakest precondition calculus of Dijkstra [9]. An event consists of a guard and a body. When the guards of an event are true, the event can be enabled. When the guards of several events are true, the choice of the triggered event is nondeterministic. In addition, a CONTEXT can be defined to specify static data of sets, constants and their axioms. An Event-B model SEES at least one context. Proof obligations are generated to ensure the consistency of the model, i.e. the preservation of the invariant by the events.

Refinement A refinement process is used to progress towards implementation. The abstract model is transformed into a more concrete and elaborated model. New variables can be introduced and the old variables can be refined to more concrete ones. This is reflected in the substitutions of the events as well. A WITH clause expresses the link between the parameters of an abstract event, (possibly removed in the refined event) and their concretization. New events may also be introduced in the refinements. These new events should not prevent forever the old ones from being triggered. A VARIANT can be introduced to ensure this property. It consists of a natural number which must decrease each time a new event is fired. Furthermore, one abstract event can be refined by several events, as well as several events can be merged into a single one. Proof obligations ensure that the refined model is consistent, i.e. its INVARIANT is preserved, and the VARIANT is decreased by the new events. Furthermore, they ensure that the refinement is correct, i.e. the refined events do not contradict their abstract counterpart.

2.2 Animation and Brama

Brama [18] is an animator for Event-B specifications. It is an Eclipse based plug-in for the Event-B platform RODIN which can be used in two complementary modes. Either Brama can be manually controled from within the RODIN or it can also be connected to a Flash graphical animation through a communication server; it then acts as the engine which controls the graphical effects.

The figure 1 shows the "classic" interface of the animation. On the left hand side, the events of the animated machine appear. They are in one of two states: enabled or disabled, depending upon the evaluation of the guards to TRUE and FALSE respectively. On the right hand side, the actual values of the machine variables are displayed. The buttons can be used to customize the display or to activate specialized value editors.

The basic user action is to click on an enabled event. This triggers three internal steps:

- to pick the values that make the guards true. When several values are possible, the choice is non deterministic;
- to compute the ACTION part of the event. If several values are possible, the choice is non deterministic;
- to re-evaluate the guards of all events.

[3] http://rodin-b-sharp.sourceforge.net

At all steps, Brama checks that the values, either provided by the user or computed by the events, do not break the invariants of the machine or the axioms in the contexts.

In a specification which includes several refinements, each one can be animated independently. This feature has two consequences. The first one is that highly non-deterministic machines which are often found at the initial steps of the specification process may not be animatable, but this does not prevent the animation of further refinements where the non-determinism has been lowered. The second consequence is that the "refine" feature of all events must be turned to `false` even if they do not change[4].

An animation session begins by setting the values of the constants in the different contexts seen (either directly or transitively) by the animated machine.

Then, the user must fire the `INITIALISATION` event, which is, at that time, the only enabled event. After this, the user will play the animation by firing the events until there is no more enabled event, or the system enters to a steady loop, or an error occurs (broken invariant or non computable action typically), or a deviation from the intended behavior is observed. In the last two cases, the specifier must go back to the specification in order to correct it.

A graphical interface can be connected to Brama in the form of a Flash[5] application. Events can be fired from the graphical interface. A mechanism of observers is provided. Expressions and predicates can be individually monitored and their value communicated to the Flash program each time it changes. Last, a scheduler mechanisms allows for the automatic firing of events.

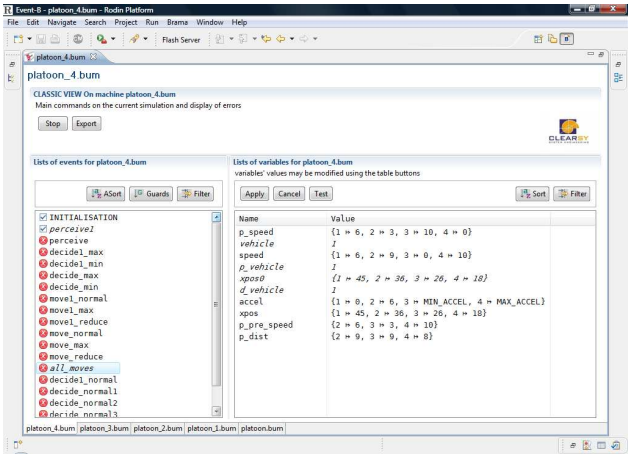

Figure 1. The Brama animator for RODIN

[4]This RODIN feature simulates a kind of inheritance from the refined machine when events are not modified in refinements.

[5]Flash is a registered trademark of Adobe Systems Inc.

3 Transformation heuristics

Animation by nature heavily depends on tools. Any limitation of the tool will be a restriction on the class of animatable specifications. To validate a specification which does not belong to this class, we need to "bring it in." We do this by applying transformation rules which are designed to keep the behavior unaltered, possibly at the expense of other properties.

While it would be interesting for the theoretician to know whether some tools' limitations come from implementation features or have a deep mathematical reason; we, as practitioners, are more interested in designing practical rules for one particular tool. However, it is important to have an explicit rule design technique so that the current effort can be leveraged and transposed to other tools.

This section first discusses the technical issues associated with animating an event-B specification with Brama. One of the important issues is the identification of the feature of the language that require transformation. Then we present the designed process to address these issues with its rationale. We insist more on rigor than on pure formality. In particular, a systematic pattern to describe transformations is presented. The last subsection shows selected transformations.

3.1 Animatable vs non-animatable

The first observation we made when trying to animate a specification was the distinction between a provable specification and and animatable specification:

1. a correct specification may not be animatable,
2. an incorrect specification may be animatable,
3. most well written specifications are likely to be non animatable.

The first two sentences were a consequence of the first error message one is likely to encounter: "Brama does not support finite axioms." Since these axioms are mandatory to discharge the well-formedness proof-obligations generated when using carrier sets, the case was settled. Beyond the anecdote (removing such technical axioms do not change the essence of the specification), this feature of Brama gave us the essential insight to dissociate proofs and animations. We could then focus on transformation rules which preserve behavior without bothering about preserving proofs (or provability).

Of course, by putting proofs aside, we are at risk of generating incorrect specifications. In fact, we can no more prove *within the formal B rules* that a transformation is correct. This implies that the correction of the transformations must be asserted through other means. We have then chosen

to follow the mathematical tradition of providing rigorous and convincing arguments as proof of the preservation of behavior for each transformation rule.

The situations where Brama cannot animate a specification can be arranged in a typology of five typical cases:

I Brama does not support the finite clause in axioms

II Brama must interpret quantifications as iterations

- **II.1** Brama only operates on finite sets
- **II.2** Brama cannot compute finite sets defined in comprehension with nested quantification
- **II.3** Brama explicitly requires typing information of all those sets over which iteration is performed in an axiom

III Brama cannot compute dynamic functional bindings in substitutions

- **III.1** Brama does not support dynamic mapping of variables in substitutions
- **III.2** Brama does not support dynamic function computation in substitutions

IV Brama does not compute arbitrary functions

- **IV.1** Functions with analytical definitions in context cannot be computed in events
- **IV.2** Functions using case analysis can not be expressed in a single event
- **IV.3** Invariants based on function computations can not be evaluated

V Brama has limited communication with its external graphical animation environment

For each situation, we have defined a "heuristic" to transform the original specification into one that can be animated. The heuristics are described following a rigid pattern:
Symptom: what reveals the situation, e.g., an error message from Brama
Transform: the expression schema in the original specification and its transformed counterpart
Caution: description of the applicability conditions, of the possible effects, and of the precautions to follow
Justification: a rigorous argument about the validity of the transformation.

This rigorous and clear description frame, although not formal, allows us to use safely animation to validate specifications because it is combined with the explicit process described hereafter.

3.2 Step-wise validation process

At the verification level, the consistency of refinement-based development processes is guaranteed by the generation of proof obligations and their discharge. Since animation requires us to loosen the provability constraint, the relation between verification and validation at the refinement level becomes an issue.

Our position is that there is no point in validating a specification which could not be verified! Such a specification is a dead-end as far as formal development is concerned.

A *verified* specification must be the starting point of the animation process. The application of the heuristics will "downgrade" it to a non provable specification. Running the animation may uncover some mistakes. These entail the modification of the *initial* specification, which then must be verified, and transformed again for proceeding with the validation. This is summed-up in figure 2.

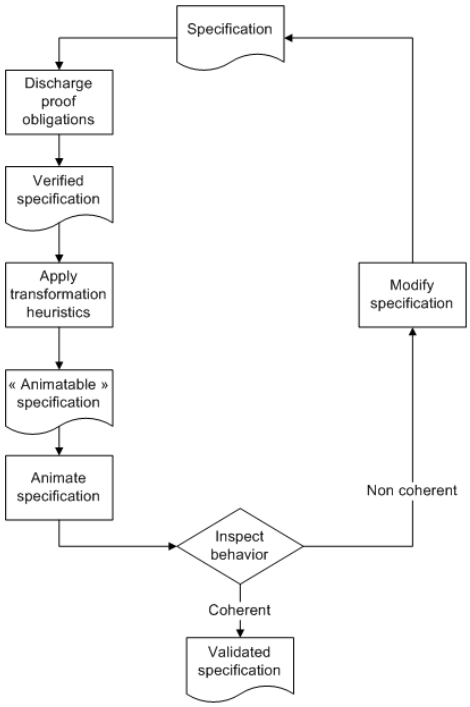

Figure 2. Step-wise validation process

It is important to note that the order between verification and validation is the reverse of what a development relying on tests would use. In the later case, there is no point in engaging a costly series of tests on a piece of code which does not fulfill users' needs.

We give verification preeminence over validation for two reasons. First, it provides us with a reasonable safeguard. Second, and more importantly, it allows us to justify some heuristics with sound arguments.

For instance, let us consider two heuristics. One calls for the erasure of an invariant (IV.3). This is safe because (1) invariant do not modify behaviors (they are only observed) and (2) proof-obligations related to maintaining the invariant have been successfully discharged. Another heuristic calls for the replacement of a set defined through complex properties of its elements by a simpler super-set (II.2). Provided we exert great care when feeding the animation with values which conform to the "complex" set, the transformation is safe because proof obligations have been discharged under the assumption that the values belonged to the "complex" set, and (1) either the values are only used (they are constants), and so properties are trivially maintained, or (2) the values are computed, but then at least one of the discharged proof-obligation was about the belonging of the computed value to the "complex" set.

Though less direct, the justification for the other heuristics rely heavily on the fact that they are applied to verified texts.

3.3 Heuristics

As can be expected from the aforementioned typology, we have designed 10 heuristics: one per category/case. The list is not closed; we may encounter in the future specifications with un-animatable traits not covered in this list.

Due to space limit, we present and discuss only heuristics II.2, III.2, IV.1, and IV.2. Interested readers are referred to [14] for a detailed presentation of I to III.2. Heuristic IV.3 calls for erasing an invariant; its most important features were discussed in previous subsection. Heuristics V is only about the introduction of "observation" variables. They are required by the communication protocol between Brama and Flash which can pass only integers: lists or function must be "broken" down.

3.3.1 Heuristic II.2: Generalize list expression

Symptom: Error message about the impossibility to build the iterators of the predicate.
Transform: Replace by a super-set

Original var = {x | ∃ n . n ∈ N1 ∧ x ∈ 1 .. n → y}
Transformed var ∈ ℙ (N ⇸ y)

Caution: This transformation loosens the constraints on the values, some maybe essential to the behavior (for instance, the property that all integer between 1 and the length of the sequence belong to the domain of the function). Brama cannot ensure anymore that the properties hold. The burden of the check is passed onto the input of the values.
Justification: Since the modified specification accepts more values than the initial specification, it has more behaviors. On the subset of values shared by the specification (that is, those values respecting the constraints left out by the generalization), both specifications must have the same behavior. Two cases must be considered:

- the value is a constant: it does not change during the animation and it keeps its properties,
- the value is a variable: at least one of the proof obligations in the initial specification deals with proving that the result of the computation belongs to the set. Since the initial specification is verified, the values in the modified specification have the same property.

This heuristic is quite specific and motivated by the absence of data-structures such as lists in event-B. Redefining ad-hoc lists is not difficult but leads to intricate expressions. It should be noted that the problem does not come from the infinite set N1, but from the doubly quantified structure.

3.3.2 Heuristic III.2: Avoid dynamic function computation in substitutions

Symptom: Error message: "Related invariant is broken after executing the event". Brama cannot apply a function defined by its graph in a substitution.
Transform: Rewrite the substitution to avoid function computation.

Original var := {x . x ∈ X | fun(x)}
Transformed var := {ran ({x . x ∈ X | x} ◁ fun)}

Justification: The transformation is simply a rewriting of the initial expression as a formula in set algebra. While less readable, it has the same semantics.

One may wonder if this heuristic could not be replaced to a simple advice: "do not use function computation in set definition!" We do not think so. To our taste, the transformed text is far less readable, hence, more difficult to understand, to use in proofs, to maintain, or to correct. This question will be discussed in section 5.2.

3.3.3 Heuristic IV.1: Inline the functions defined in contexts in events

Symptom: Functions defined analytically as constants in contexts can neither be initialized nor computed in events.
Transform: Substitute function calls by their "inlined" equivalent

Original (in Context)	∀x. x∈S ⇒ f(x) = expression(x)
Original (in Event)	f(v)
Transformed (in Context)	true
Transformed (in Event)	expression(v)

Caution All occurrences of f in the specification must be replaced; special care must be exerted replacing formal parameters by actual values.
Justification: In a mathematical context, the value f(v) is equal to its definition expression where v has been substituted to x; both expressions are interchangeable.

Contexts in Event-B are precisely meant to contain constants and general definitions, such as functions. Using this structure eases the proofs and provides better legibility. As for III.2, the "inlining" heuristic is strongly connected to the issue of readability and understandability of formal texts.

3.3.4 Heuristic IV.2: Replicate events which use functions defined "by cases"

Symptom: Same as IV.1, plus a function defined "by cases"
Transform:

```
Original (in Context) ∀x. x∈S ⇒ (p(x) ⇒ f(x) = expression(x) ∧
                                 q(x) ⇒ f(x) = expression'(x))
Original (in Machine) EVENTS
                        EVENT A
                        WHEN ...f(v)...
                        THEN ...f(v)...
Transformed (in Context) true
Transformed (in Machine) EVENTS
                           EVENT A1
                           WHEN ...
                             grdc1 p(v)
                           THEN ...

                           EVENT A2
                           WHEN ...
                             grdc1 q(v)
                           THEN ...
```

Caution: This heuristic must be followed by the application of IV.1. Check that all cases have been covered. Be particularly careful if the function is applied to several, different actual parameters; this may require several application of this heuristic.
Justification: The predicates used in the "by case" definitions are equivalent to guards in events. They have the same form and are used for the same purpose. Events A1 and A2 are copies of A, except for the new guard: their union is equivalent to A. Hence the transformed specification has the same behavior as the initial specification.

This heuristic entails major surgery in the specification. A blind application may introduce many copies of the events. By using the structures of the other guards (some may already prevent cases in the function definition to be used) and by grouping several function into one transformation, it is possible to reduce the number of duplications.

4 Case studies and lessons

This section describes two case studies which were the incentive for this work. They also provide us with a good test field as they contain most of the features that can be expected from event-B specifications.

Both specifications concern the domain of land transport systems. They are part of cooperative projects. TACOS, supported by ANR, is an effort to integrate components and non functional properties into formal requirement specifications. In particular, safety critical properties must be assessed and formalized. CRISTAL, supported by the *Pôle de compétitivité Alsace–Franche-Comté*, is a joint project with the industrial goal of designing urban mobility systems based on autonomous vehicles. As these systems interact with humans and operate on public space, the certification issue is a major problem.

Formal methods have already been used successfully in transport domain, mainly for rail control systems, such as the Roissy VAL [3]. Dealing with new urban transport systems introduces a concern: their integration into already existing systems. So a specification must describe behaviors consistent with the existing world, hence, igniting the importance of early validation.

4.1 Transport domain model

The specification in this case study is about the modeling of the land-transportation domain. In the model, we want to express properties that any system working within the domain is expected to meet and maintain.

In this specification effort, the focus is on the formal definition of concepts, constraints and properties, rather than on the implementation of a particular system. Refinement is then used to introduce new notions; the proof obligations serve to guarantee the consistency of the model.

As our intuitive model is the road system, it is essential that the formal description be kept consistent with observed, or desired, behaviors. Ideally, each refinement should be validated.

The current specification consists of 8 refinements. It is organized into five abstraction levels:

1. definition of the network and the notion of travel. The most interesting part lies in the contexts which describe the abstract topology of a network: connections, hubs, junctions, stations, etc. A travel is simply defined as moving from one station to another.

2. definition of travel constrained by the topology of the network. Vehicles travel by following a sequence of paths linking the departure station to the destination. Notions of paths and routes are introduced.

3. definition of interaction between moving vehicles at intersections. The absence of collisions at intersections is an essential property of a transport system. This is abstractly modeled by the constraint that vehicles may enter hubs only if they can carry them. A small protocol, including a wait event, is introduced to model the crossing of a hub.

4. definition of the travel-time. These refinements introduce a first simplified concept of duration and model its computation.

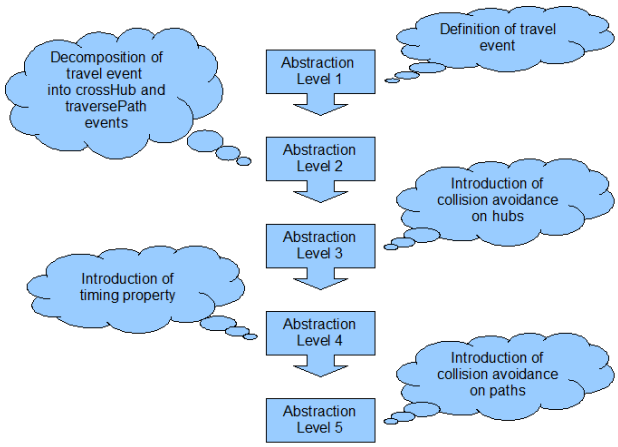

Figure 3. Level of abstractions of transport domain model

5. introduction of kinematic movement along paths. This refines the notion of time toward a continuous model (although still discretely defined); it introduces also the constraint of absence of rear-end collisions.

Figure 3 summarizes the different levels of abstraction of transportation domain model. A detailed description of the model can be found in [13].

This specification exhibits several properties which call for animation, namely:

- complex data which constraint behaviors (following a route),
- protocols and iterations (travel as sequence of stages, hub crossing protocol), and
- non deterministic interaction between elements (autonomous vehicles).

Intuitively, these properties mean that the firing of events must follow some ordering rules, but this cannot be specified in Event-B. The language is not to blame, as this feature is implied by the mathematical underlying theory. Even the notion of `VARIANT` included in Event-B was not useful for us as soon as the notion of waiting was introduced.

Until the fourth level, only heuristics from the categories I to III are needed. Then, setting up animations was easy and we used them intensively. Actually, we used animation more as prototyping rather than validation while beginning work on refinements. It helped us understand and fix desired behaviors.

The most important difficulty we experienced was of a very practical nature. Entering values to run an animation is difficult in the current implementation of Brama:

- Brama uses only integers, so cumbersome coding is often needed,
- Brama uses the same strict mathematical syntax as Event-B; input are then long, tedious to type, difficult to read, and prone to typing errors, and
- the value editors are of the most basic nature, not even supporting copy/paste.

The first item is more an annoyance than a real problem, in particular when animations are not connected to a graphical interface. Combined, the last two items are a real problem. Their major effect is to limit animations to small scale experiments. Tools to help with value generation and management are much needed.

Not all refinements were animated, but we made sure to have one for each level. A typical example of a refinement that was not animated was the sixth one, where the notion of time was introduced. The aim of the refinement was mainly to set the vocabulary (`travel_time`, `clock`) and very general properties: time always increases, clock ticks, etc. The definitions are highly non deterministic and there is not much in term of new behaviors. Animation of that particular refinement would not bring much information. The seventh refinement, where the actual computation of time—a new complex behavior—is defined, was subjected to animation.

The interesting point to note is that a machine can be animated while its refinement may not. There is no monotonicity and, for this particular study, this was a good thing.

Detailed stepwise animation of each refinement step of this case study can be found in [14].

4.2 Situated multi-agent platooning system

The second case study deals with a specification of *platooning*. Platooning is a mode of moving where vehicles are synchronized and follow one another closely. A platoon can be seen as a road-train where cars are linked by software instead of hardware. Platooning has several potential uses in an urban mobility system: augmenting throughput, "herding" unused cars to stations, or running "transient" buses for instance. A key issue is the certification of platooning.

Several platooning control systems are being developed and experimented. One locally developed is based on situated multi-agent (SMA) theory. Each car has its own local control algorithm which uses a perception/decision/action loop; the platooning behavior is an emerging property [8, 16].

An Event-B specification of the local model has been written [10, 6, 7] as an effort to make it amenable to the formal techniques required by certification.

Contrary to the previous case study, the structure of the development can be interpreted as a sequence of refinements toward an implementation. Each refinement decomposes some events to make explicit a part of the general computation.

The specification consists of five machines (four refinements):

Platoon defines platoons and sets the basic safety property. It contains only one event, all_move, where all vehicles change positions while keeping safe distance.

Platoon_1 decomposes the event into one which move the leader vehicle and one which moves the followers. This organizes the basic "iteration along the platoon" of each move.

Platoon_2 computes the length of each basic move. This leads to introduce in the contexts kinematic functions and to refine the move events into several ones corresponding to different situations (maximum and minimum speed reached or not). This models the *action* part of the SMA.

Platoon_3 introduces the notion of *decision* of the SMA model into the specification. Two events, one for the leader, one for the followers, are introduced and integrated in the control loop.

Platoon_4 introduces the notion of *perception* which allows decision events to be refined so the actual computation of the parameters of the control law (acceleration) can be performed.

Although the last refinement is very close to an implementation, in spirit if not in form, yet we decided to use animation to validate the specification for several reasons. The first was curiosity as the heavy use of functions was challenging, the second was to compare the results of the animation with the results of simulations that had been previously made, and the last was to confirm that a certain "formal approximation" was legitimate.

The last reason is a consequence of using discrete tools to model what is inherently continuous. In this case, all proof-obligations were discharged, assuming one property, namely $x(y/z) = (xy)/z$, holds. True in $\mathbb{R}$, this property is false in $\mathbb{N}$. However, the difference becomes actually negligible when numerators are much bigger than denominators. Animation with realistic values gives insight on the validity of the "approximation" and on the solidity of the model.

Although all machines have been animated, the first four are not particularly interesting. The non deterministic definition of the parameter of the control law does not allow for long automatic run of the animation. To observe interesting behaviors, we have to feed "coherent" values to each event which is fired. This can be useful for a quick look into the behavior, but not much more. The interesting animation was for `Platoon_4`.

However, following the refinement structure provided us with a better grasp of the complex transformations need. The first complex modification came with the second refinement. Context contains definition of new_xpos function as

```
∀ xpos0,speed0,accel0 .
  (
    (xpos0 ∈ NAT ∧ speed0 ∈ 0..AX_SPEED ∧ accel0 ∈ MIN_ACCEL..AX_ACCEL)
     ⇒
    (new_xpos(xpos0 ↦ speed0↦accel0) = xpos0 + speed0 + (accel0 / 2))
)
```

which models a kinematic law. It was used in some event guards in the form

```
nxpos = new_xpos(xpos(vehicle) ↦ speed(vehicle) ↦ magic_accel)
```

Using heuristic IV.1, we rewrote the guards as

```
nxpos = xpos(vehicle) + speed(vehicle) + (  magic_accel/2))
```

The most important complication came with another kinematic function new_xpos_max which is quite similar to new_xpos, except there is a case definition:

```
∀ xpos0,speed0,accel0 .
  (
    (xpos0 ∈ NAT ∧ speed0 ∈ 0..AX_SPEED ∧ accel0 ∈ MIN_ACCEL..AX_ACCEL)
     ⇒
      (accel0 = 0   ⇒ new_xpos_max(xpos0 ↦ speed0↦accel0) =
            xpos0 + MAX_SPEED)
      ∧
       (accel ≠ 0 ⇒ new_xpos_max(xpos0 ↦ speed0↦accel0) =
          xpos0+MAX_SPEED − (((MAX_SPEED−speed0)∗(MAX_SPEED−speed0))/
                                  (2  /accel0))
       )
)
```

The events using the new_xpos_max function had to be duplicated, one with the guard accel=0 and the other with its negation.

The same two heuristics had to be applied several times. The trickiest situation was actualy the two guards

```
grd5  naccel=new_accel(p_dist(d_vehicle)↦p_speed(d_vehicle)↦p_pre_speed(d_vehicle))
grd3  ∃ g1,g2 .
 (g1 ∈ {new_xpos, new_xpos_max, new_xpos_min} ∧ g2 ∈ {new_xpos,
   new_xpos_max, new_xpos_min} ∧
   (g1(xpos(d_vehicle−1) ↦ speed(d_vehicle−1) ↦ accel(d_vehicle−1))
    − g2(xpos(d_vehicle) |− speed(d_vehicle−1) ↦ naccel) > CRITICAL_DISTANCE)
 )
```

Out of three functions used in grd3, two are defined by case but all are used two times with different arguments. We were lucky enough that the cases for the function are the same (acceleration null or not) so we had only four duplications (cases for accel(d_vehicle−1) and naccel). After application of the transformations, the number of event in Platoon_4 went from 15 to 20.

The last step was to set-up a small graphical interface in Flash so we could have a synthetic view of the moving platoon. Technically, we had to introduce a new refinement so that "observation" variables could be set. The reason comes from the limited data types that Brama currently communicate with the Flash server: integers and boolean. As the

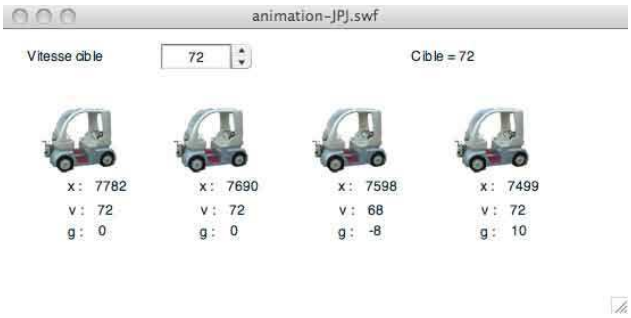

Figure 4. Animation of platoons

model uses discrete functions to record current information of the vehicles, we had to "split" them into different variables. They are all concentrated in the move_all event. The end result is shown by figure 4.

As can be seen on the interface, the cruising speed of the platoon can be controled. Setting this control required us to modify the specification in which the cruise speed was initially defined as a constant. From our point of view, the initial specification was unrealistic in this respect and the animation allowed us to spot this, small, inconsistency.

5 Conclusion

The previous sections have presented a set of techniques and a process to make validation a concurrent activity to the development of a formal specification. While effective, they raise several questions about the soundness of such validation, about the possibility to avoid transformation, about their relation to refinement-based development processes, and about the tool adequacy. We propose a few answers here.

5.1 Correctness of heuristics

The soundness of our approach depends critically on the correctness of the heuristics. As we have seen, the heuristics are not formal semantic-preserving transformations in a strict sense since most do not preserve the provability of the specification. However, they can be said to be correct with respect to the validation if they preserve the behavior.

The issue of designing fully automated and formal rules to transform the initial specification into a provable and animatable one is still open. Apart from the fact that we doubt the possibility to achieve such a goal, we do not think it is a good idea for two reasons. The first reason is practical. Heuristic II.1 shows that a good understanding of the domain is necessary to limit iterations to a practical range. Similar to program testing, the point is rapid exploration of domain to uncover problems. So, a slow animation would make the process unpractical. The second reason lies with the issue of correcting the errors. Heuristic II.2 shows that transformations would entail drastic changes in the specification, maybe even in its structure. The problem of tracing a behavioral error back to the initial specification is likely to become very complex. For the same reason, the use of the heuristics form category IV requires the intelligence of the developer. Blind applications of the transformation will multiply events to a huge number. Again, identifying mistakes would become a much more complex problem.

We think that our pragmatic approach, based on a rigorous process and rigorous arguments is more appropriate to our aim.

5.2 Writing animatable specifications

A way to avoid the hassle of transformations would be to write specifications which are animatable form the start. Though appealing, we do not think this idea is effective.

If we try to write an animatable specification, we must introduce very early in the design process some arbitrary constraints (cf heuristic II.1) and we must use convoluted expressions (cf heuristic III.2). Even worse, heuristics IV shows that we must avoid the elegance of function definitions and replace them by long clumsy expressions. In all situations, the specifier commits a sin: over specification, esoteric notations, and unreadable text. The advice given for programming to keep things simple, general, and readable holds true for specification as well. More errors were corrected during the elaboration of the specifications while discharging the proof obligations and careful cross-reading than during the animations. Of course, they were of different nature.

Good readability and elegance are key factors for the general acceptance of formal methods. Animation is just a technical activity which should not impose constraints on the specification.

5.3 Relation to refinement-based development processes

Not every refinement step is animatable, but this is not inconsistent with using animation as a kind of quality-assurance activity during development. As stated in [13], a development can be structured into abstract levels; one animation per level is sufficient. In fact, the first refinement of a level may often have a non-determinism too wide to allow for meaningful animation (concept introduction), but subsequent refinements get the definitions of the new concept precise enough to allow animation.

In complement to the validation, we have found that animations can also be a great help when designing a particular refinement step. In such situation, animation acts more like

a prototype than like a validation tool. Even before the new refinement is proven, it is possible to use the heuristics to make a "quick and crude" animations. This allows specifiers to get a better grasp on the new behavior they want to introduce. Of course, once the refinement has been fixed, it must be proven and the rigorous validation process must be carried out.

5.4 The missing tools

If we want step-wise animation to become a routine tool for specification developers, there is a strong call for specific tools.

The Brama animation engine is actually most adequate for the task. Improvements in the communication between Brama and Flash, notably by extending the types of values that can be transmitted, would be welcome. However, two tools are sorely lacking at present.

The first would be a tool to help in generating and inputting values into the constants of the contexts. Presently, the task is tedious and error prone. If a general tool would probably be non realistic, at least a programmatic API to access and to set values from external programs should be implemented.

The second tool is the implementation of the heuristics. The major modifications implied by heuristics of category IV are difficult and tedious to carry by hand. We are currently experimenting with a specification of the platooning problem in 2D. While the animation should not require new heuristics, it is not currently clear that we will be able to manage by hand the complexity due to the increased number of functions and events. There does not seem to be theoretical or practical reasons to prevent heuristics to be implemented. A future goal is then set.

References

[1] J.-R. Abrial. *The B Book*. Cambridge University Press, 1996.

[2] J.-R. Abrial and S. Hallerstede. Refinement, Decomposition, and Instantiation of Discrete Models: Application to Event-B. *Fundamenta Informaticae*, 77(1-2):1–28, 2007.

[3] F. Badeau and A. Amelot. Using B as a high level programming language in an industrial project: Roissy VAL. In *Formal Specification and Development in Z and B*, volume 3455 of *LNCS*, pages 334–354. Springer-Verlag, 2005.

[4] R. M. Balzer, N. M. Goldman, and D. S. Wile. Operational specification as the basis for rapid prototyping. *SIGSOFT Softw. Eng. Notes*, 7(5):3–16, 1982.

[5] J. Bendisposto, M. Leuschel, O. Ligot, and M. Samia. La validation de modèles Event-B avec le plug-in ProB pour Rodin. *Technique et Science Informatiques*, 27(8):1065–1084, 2008.

[6] S. Colin, A. Lanoix, O. Kouchnarenko, and J. Souquières. Towards Validating a Platoon of Cristal Vehicles using CSP‖B. In J. Meseguer and G. Rosu, editors, *12th International Conference on Algebraic Methodology and Software Technology (AMAST 2008)*, number 5140 in LNCS, pages 139–144. Springer-Verlag, July 2008.

[7] S. Colin, A. Lanoix, O. Kouchnarenko, and J. Souquières. Using CSP‖B Components: Application to a Platoon of Vehicles. In *13th International ERCIM Wokshop on Formal Methods for Industrial Critical Systems (FMCSCS 2008)*, LNCS. Springer-Verlag, Sept. 2008.

[8] P. Daviet and M. Parent. Longitudinal and lateral servoing of vehicles in a platoon. In *Proceeding of the IEEE Intelligent Vehicles Symposium*, pages 41–46, 1996.

[9] E. W. Dijkstra. *A Discipline of Programming*. Prentice Hall, 1976.

[10] A. Lanoix. Event-B specification of a situated multi-agent system: Study of a platoon of vehicles. In *2nd IFIP/IEEE International Symposium on Theoretical Aspects of Software Engineering (TASE)*, pages 297–304. IEEE Computer Society, 2008.

[11] M. Leuschel, L. Adhianto, M. Butler, C. Ferreira, and L. Mikhailov. Animation and model checking of CSP and B using prolog technology. In *Proceedings of the ACM Sigplan Workshop on Verification and Computational Logic VCL'2001*, pages 97–109, 2001.

[12] M. Leuschel and M. Butler. ProB: A model checker for B. In K. Araki, S. Gnesi, and D. Mandrioli, editors, *FME 2003: Formal Methods*, LNCS 2805, pages 855–874. Springer-Verlag, 2003.

[13] A. Mashkoor, J.-P. Jacquot, and J. Souquières. B événementiel pour la modélisation du domaine: application au transport. In *Approches Formelles dans l'Assistance au Développement de Logiciels (AFADL'2009)*, page 19, Toulouse France, 2009.

[14] A. Mashkoor, J.-P. Jacquot, and J. Souquières. Transformation Heuristics for Formal Requirements Validation by Animation. In *2nd International Workshop on the Certification of Safety-Critical Software Controlled Systems (SAFECERT 2009)*, page 16 pp., York (UK), 2009.

[15] Esterel Technologies: SCADE Language Reference Manual, 2004.

[16] A. Scheuer, O. Simonin, and F. Charpillet. Safe Longitudinal Platoons of Vehicles without Communication. Research Report RR-6741, INRIA, 2008.

[17] R. Schmid, J. Ryser, S. Berner, M. Glinz, R. Reutemann, and E. Fahr. A survey of simulation tools for requirements engineering. Technical report, 2000.

[18] T. Servat. BRAMA: A New Graphic Animation Tool for B Models. In *B 2007: Formal Specification and Development in B*, pages 274–276. Springer-Verlag, 2006.

[19] J. I. Siddiqi, I. C. Morrey, C. R. Roast, and M. B. Ozcan. Towards quality requirements via animated formal specifications. *Ann. Softw. Eng.*, 3:131–155, 1997.

[20] H. T. Van, A. van Lamsweerde, P. Massonet, and C. Ponsard. Goal-oriented requirements animation. In *RE '04: Proceedings of the Requirements Engineering Conference, 12th IEEE International*, pages 218–228, Washington, DC, USA, 2004. IEEE Computer Society.